Profiling and Performance

Gabriele Svelto, Thomas Zimmermann

June 2, 2013

Algorithms and Complexity

Computational Complexity

- Given
  - Algorithm
  - Input of length n
- How many steps are necessary to complete algorithm as $n \to \infty$?
- Big-O notation
- $algorithm(n) = O(steps(n))$ as $n \to \infty$

Typical Complexity Classes

$O(1)$ constant complexity, sign function, absolute values, searching in well-tuned hash tables

$O(logn)$ logarithmic complexity, binary searches, balanced search trees

$O(n)$ linear complexity, linear searching

$O(nlogn)$ linearithmic complexity, building search trees

$O(n^k)$ polynomial complexity, naive sorting (e.g, bubble sort), matrix multiplication

$O(k^n)$ exponential complexity, traveling salesman problem

- example `sort.c`

Typical Complexity Classes

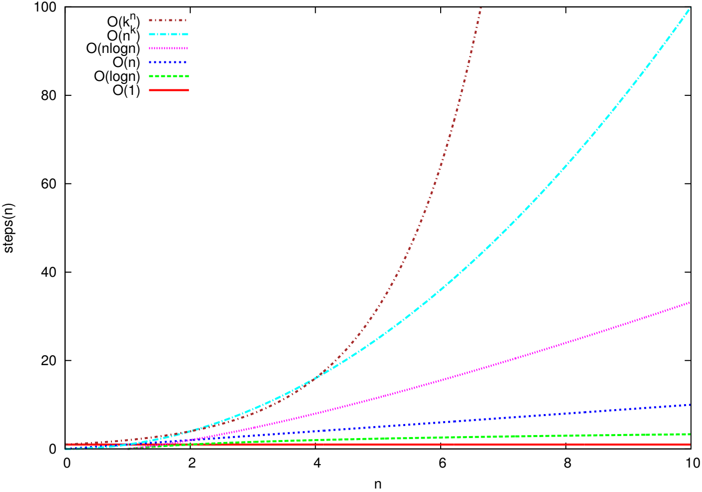

Figure: Complexity Classes

Determining Complexity

- use standard algorithms with known complexity

or

- try to describe relation between n and number of primitive operations
- example of bubble sort
  1. n iterations
  2. first iteration: $n - 1$ compare operations
  3. second iteration: $n - 2$ compare operations
  4. $n'th$ iteration: $n - n = 0$ compare operations
  5. average per iteration: $n/2$ compare operations
  6. overall complexity: $O(n * n/2) = O(n^2/2) = O(n^2)$
- combinations of algorithms have the maximum complexity of primitive algorithms
- example of bubble-sorting absolute values
  1. walk over all elements ($O(n)$)
     - compute absolute value for each element ($O(1)$)
  2. bubble-sort the results ($O(n^2)$)
  3. overall complexity: $O(n * 1 + n^2) = O(n^2)$

Which Algorithm Is Best?

- naive answer: use algorithm with lowest complexity
- but there are exceptions
  - *n* is small
  - better algorithms come with setup costs (e.g., binary searches need sorted input)
  - hardware has branch prediction and is optimized for linear memory access (by prefetching memory)
    - binary searches hop around in elements
    - linear searches will walk over elements
- also consider other resources

- *B. Kernighan, R. Pike: The Practice of Programming, Addison-Wesley 1999*

JavaScript

- General tips
  - Don't mix types
  - Use simple types as much as possible
    - Use an integer as an ID instead of a string
    - Use short arrays to store short vectors (e.g. x, y, z coordinates)
  - Recompute simple values, don't store them
  - Use arrays when possible

JavaScript (continued)

- DOM navigation
  - Use `node.children` not `node.childNodes` to navigate child nodes
  - Always iterate at the same level with `nextElementSibling`
- Object management
  - Use standard objects over classes and prototypes
  - Don't add new properties to an object after initialization
  - Don't remove properties with `delete`

Garbage collection

- Use small types whenever possible
- Avoid creating too many temporary objects
- Don't hold objects you don't need any more
- Watch out for variables held by closures
- Again, don't add new properties to an object after initialization
- Again, don't remove properties with `delete`
- Unbind all unused listeners
- If you're keeping a cache around or a similar structure listen to `memory-pressure` events and flush it when you receive them
- Be careful when manipulating strings
  - Avoid useless concatenations / splits
  - Avoid concatenating to large strings

CSS

- Keep selectors simple
- Complex selectors can be expensive and make your styles hard to understand for people reading the code
- Use ID-, tag- and class-based rules
  - `#toppanel {...}`
  - `.squarebutton {...}`
  - `a {...}`
- Avoid universal selectors
  - `[hidden=true] {...}`
- In general the less elements a rule can apply to the better

Layout

- Always specify sizes for elements if possible
- Prefer CSS backgrounds to image tags
- Setting a position / size property will likely trigger a reflow, group those changes to multiple elements to avoid causing more than one
- Reading a position / size property before the page has been reflowed will cause a reflow and it will be a *synchronous* one!
- Use a `DocumentFragment` to append elements to a DOM tree
- Fully initialize a new element before adding it to the DOM tree

Painting

- Group methods that do or cause repainting
- Avoid animated images (PNG/GIF), are expensive to paint and inflexible
- Make good use of `<canvas>` elements
  - For animation that is not possible via CSS properties
  - For drawing small animated UI elements (e.g. status icons)
  - For animated images
  - They are `requestAnimationFrame()`-friendly
  - Use native types in the drawing code (arrays, etc...)
  - Do not use them for things that can be done using conventional methods, native Gecko is faster at drawing than JavaScript code

Startup performance

- Don't include scripts or stylesheets that are not immediately needed, load them when needed
- Use the "defer" or "async" attribute on script tags needed at startup
- Create DOM elements only when they are actually needed
  - An element can be hidden in a comment and extracted from it when needed
  - `<div id="foo"><!-- <div> ... --></div>`
  - `foo.innerHTML = foo.firstChild.nodeValue`
- Optimize your assets
- Don't wait for storage / remote resources, load them while the application is already running

Application responsiveness

- Can be hampered by a number of issues
  - Blocking on slow operations (I/O, network)
  - Long-running CPU-intensive operations
  - Excessive updates / refreshes
  - Platform limitations
- Use asynchronous APIs as much as possible
  - For storage I/O
    - AsyncStorage is what you want
    - Keep away from LocalStorage (unless it's used in a worker)
    - Using IndexedDB directly is fine but keep everything asynchronous
  - For network resources
    - You don't want your application to wait for a timeout to expire
  - For local resources too opt for asynchronous interfaces when both async and sync are available

Application responsiveness (CPU usage)

- Limiting CPU usage is always a win
  - Lets other applications run smoothly
  - Makes the CPU available for background tasks
  - Lengthens battery life!
- Use web worker threads to offload CPU-intensive or long-running tasks
  - `https://developer.mozilla.org/en-US/docs/DOM/Using_web_workers`
  - Keeps the main thread free and thus the application responsive
  - Limitation: a worker cannot manipulate the DOM!
- Watch out for event handler spam
  - Data transfer progress updates
  - Rapidly firing timers
  - Throttle or group events when possible
- Always use requestAnimationFrame() for animations

C and C++ - Executables and Address Spaces

- executables consist of shareable sections

  .text program code
  .rodata read-only data

- and non-shareable sections

  .data write-able data
  .bss zero-initialized writeable data

Figure: Memory layout

- address spaces are composed of executable's sections plus
  - stack
  - heap memory

C and C++ - Static Memory Allocation

- example `lut.c`
- maximize page sharing among running programs
  - init static data with zero to store them in **.bss** segment
  - mark constant data as `static const` to put it into read-only sections
- minimize dynamic relocations
  - performed by dynamic linker when loading program
  - computes runtime-adresses of static data and updates references
  - updated references are not sharable among programs
  - avoid indirections
    - static pointers to static pointers
    - prefer `static const str[] = ""` over `static const *str = ""`
- *J. R. Levine: Linkers & Loaders, Academic Press 2000*
- *U. Drepper: How To Write Shared Libraries, http://www.akkadia.org/drepper/dsohowto.pdf*

C and C++ - Dynamic Memory Allocation

- memory allocators organize memory in larger segment
- unused segments of the same size are maintained in the same data structure (list, tree, etc.)
- allocation requires lookup of free segment from the data structure
- overhead from search operation

C and C++ - Dynamic Memory Allocation

- example `concat.c`
- reuse allocated memory if possible
  - prevents expensive `malloc`/`free` cycles
- similar behavior in C++, but extra costs from (de-)construction
  - don't call `new`/`delete`
  - reuse existing instances (e.g., strings)
  - reset object state and fill with new data

C and C++ - Word-sized Data

- example `concat.c`
- use word-sized data streams to optimize number of load operations
- if possible
  - prefer `mem*` over `str*`
  - prefer `float` over `double`

C and C++ - Other Tips

- initialize class members in constructor, don't assign
- use references or pointers for passing objects to functions
- use references or pointers for returning class members
- construct objects in return statement to enable return-value optimization
- overload functions for different argument types

- *S. Meyers: Effective C++, 3rd ed., Addison-Wesley 2005*

CPU - Pipelines

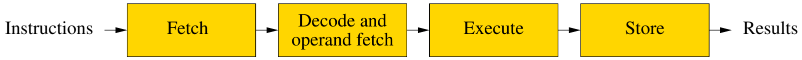

Figure: 4-stage processor pipeline

- ideally 1 instruction per pipeline per clock cycle
- need to keep the pipeline filled with instructions
- multiple next instructions possible after conditional branches

CPU - Branching

- processor tries to predice target of a conditional jump instruction
  - statically (e.g., always expect true)
  - dynamically with branch-prediction buffer
- if correct, no overhead
- otherwise
  1. processor throws away results of incorrect branch
  2. clears pipeline
  3. starts executing instructions of correct branch
- overhead of incorrect predictions depends on pipeline length (up to 20 clock cycles)

CPU - Branch-less Code

- example `sgn.c`
- advantages
  - no branch prediction necessary
  - frees slots in the branch-prediction buffer
  - often allows use of mutiple pipelines in parallel
- disadvantages
  - might require more computation
  - no expensive computation possible

CPU - Branch-less Code

- compute results without conditional jumps
- use multiply $*$ instead of logical and &&
  - && does not evaluate right-hand side if left-hand side is false
  - requires a conditional jump
  - $*$ always evaluates both sides, hence no conditional jump
- use add $+$ instead of logical or $||$
  - same as for &&
- negate twice to compute 0 or 1
  - first negation maps 0 to 1, and any other value to 0
  - second negation maps 1 back to 0, and 0 to 1
- do expensive computations beforehand, only use results
- use look-up tables for complex mappings
- *Bit Twiddling Hacks:* `http://graphics.stanford.edu/~seander/bithacks.html`

Memory - Look-up Tables

- example `lut.c`
- advantages
  - map arbitrary input to arbitrary output
  - predictable overhead
- disadvantages
  - additional overhead from memory access

Memory - Alignment

- load instructions operate along word boundaries

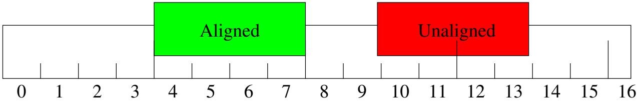

Figure: Memory access

- align data to word boundaries
  - single load instruction
  - ABI requires this
  - gcc offers `__attribute__((align(n)))`
- can result in unused bytes within data structures
  - example `align.c`
  - arrange structure fields according to alignment
  - use `pahole` for optimizing data structures

Memory - Caches

- processor fetches whole cache lines (32, 64 Byte) at once
- align larger data structures to cache-line boundary
- first-used data should go at the beginning of cache line
- keep related data on the same cache line
- use data types with minimum size
- use individual bits

Memory - False Sharing

- two global variables with unrelated data might be located on the same cache line
- processor fetches whole cache lines at once
- unmodified cache lines are **shared** by all processors
- writing to a cache line marks it as **dirty**
- affects all contained values
- other processors will update their cache line from the modified copy even if they don't operate on the modified value
- known as *False Sharing*
- try to put unrelated global data onto separate cache lines

Memory - Cache Line Bouncing

- processors use shared variables for communication with each other
  - data exchange
  - syncronization
- write operations invalidate cache lines
- read operations need to fetch cache lines
- known as *Cache Line Bouncing*
- can be avoided by good concurrency control
- *U. Drepper: What Every Programmer Should Know About Memory,* *http://www.akkadia.org/drepper/cpumemory.pdf*

Concurrency Control

- mechanism of protecting concurrent access to shared resources against each other
- aka. locks vs. atomic ops

Concurrency Control - Atomic Ops

- atomic operations modify single values
- advantages
  - good if lock contention is low
  - no dead locks
- disadvantages
  - more overhead than non-atomic operations, because of bus lock
  - no progress guaranteed (known as *live lock*)

Concurrency Control - Locking

- works on arbitrary data
- advantages
  - good if lock contention is high
  - operating-system scheduler can guarantee fairness
- disadvantages
  - more initial overhead because of system-call
  - dead locks possible
- increase lock granularity if contention is high

Tools

`readelf`	information about ELF binaries
`perf`	system-wide profiling for Linux
`oprofile`	alternative to `perf`
`pahole`	layout of data structures in memory

Profiling

- Why profiling?
  - To identify hot-spots, regressions and unpredictable issues
  - To get an accurate idea of the overall performance profile of an application and not just a part of it
  - To make informed decisions on what to optimize
- *Never optimize without profiling first!*

Profiling with the built-in profiler

- SPS built-in profiler
  - The best way to profile anything within Firefox and FxOS
  - `https://developer.mozilla.org/en-US/docs/Performance/Profiling_with_the_Built-in_Profiler`
- Advantages
  - Captures both native and JavaScript code
  - Has complementary tools for spotting events (GC, layout, etc)
  - Profile information is easy to read and share
- Disadvantages
  - Limited to the main thread
  - Limited native code analysis in FxOS
  - Granularity can be too coarse and samples can be skewed
  - Does not capture system effects
  - Requires a special build

Profiling on your device

- Configuring your build
  - Start with a regular B2G build
  - Make sure `elfhack` is disabled in `.mozconfig`
  - `ac_add_options --disable-elf-hack`
- Rebuild and flash your device
- Start the profiler with `./profile.sh start`
- Check for the running applications with `./profile.sh ps`

```
  PID Name
----- ----------------
 4989 b2g                profiler running
 5037 Usage              profiler running
 5038 Homescreen         profiler running
 5203 (Preallocated a    profiler running
```

Capturing one or more profiles

- You can capture the profile of an application by specifiying it's name or PID
  - `./profile capture Homescreen`
  - `./profile capture 5038`
- If all goes well you'll end up with a .sym file:

```
Signalling PID: 5038 Homescreen ...
Stabilizing 5038 Homescreen ...
Pulling /data/local/tmp/profile_2_5038.txt into profile
Adding symbols to profile_5038_Homescreen.txt and creat
Removing old profile files (from device) ... done
```

- You can also capture profiles for all processes at the same time by not specifying any parameter
  - `./profile capture`

Analysing your profile

- The profile will contain up to 100s of the process' activity
- Default sample time is 10ms, it can be lowered to 1ms at most
- When you have many processes open capturing a profile might kill some processes due to OOM so tread carefully
- To analyze your profile you will need to upload it to our web-based service Cleopatra
  - `http://people.mozilla.com/~bgirard/cleopatra/`
  - It can also be run locally but then you'll lose the ability to share profiles
  - The code is available here `https://github.com/mozilla/cleopatra`

Cleopatra demo

Studying a profile

- Reading call-stacks
  - JavaScript code can be easily analyzed and referenced
  - Native code currently uses markers, it's important to spot major ones
    - `nsAppShell::ProcessNextNativeEvent::Wait`
    - `JS::EvaluateString` and `js::RunScript`
    - `Timer::Fire`
    - `nsRefreshDriver::Notify`
    - `Paint::PresShell::Paint`
    - `Layout::Flush`
    - `GC::GarbageCollectNow`
- Cleopatra provides hints
  - GC markers
  - Layout markers
  - I/O markers

Studying a profile *continued*

- Filering
    - JavaScript-only filtering
    - Narrow down to a single function/method
- Inverting call stacks
- Zoom as needed to get a better idea of what's going on
- Upload your profile and add a link to your ticket for easy sharing

Limits and pitfalls

- Only the main thread is currently sampled, if you have worker threads significant time might be spent on them
- A lot of activities are delegated to the main B2G process, capturing both your app and the b2g process is often a good idea
- The profile shows real-time, not CPU time, if the phone is loaded it will appear as the app is slower
- The profile does not show system activity
  - Use `top` to spot high system activity
  - Watch out for I/O activity, check for `write` or `read` calls in your profile, use `top` to estimate the wait time
  - Use `perf` if all else fails
- IPC will not show up in the profile so be careful